Massachusetts Institute of Technology
6.892: Theory of Parallel Systems
Professor Charles E. Leiserson

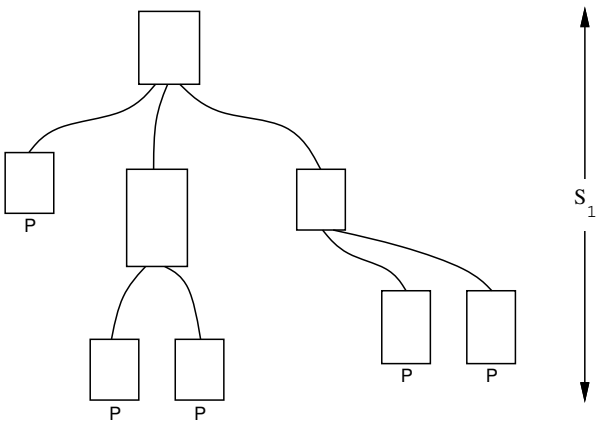
Lecture 11
March 13, 1997
Scribe: Ron Weiss

Lecture 11: Space Bounds

1 Analysis of Stack Space

Read handout 21, "An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms" (concentrate on sections 5-6).

Recall the structure of a cactus stack:



Let

S_1 = length of the deepest recursion for a single processor given a specific problem.
S_P = space used by a P processor execution of the same problem.

Cilk uses a *busy-leaves* scheduler, i.e. all the leaves are always worked on by a processor. This implies that at any time, the cactus stack has $\leq P$ leaves. Therefore, we get the following bound on space utilization:

$$S_P \leq P \cdot S_1$$

In order to maintain the busy-leaves scheduling property during work stealing:

- Whenever a processor p finishes a subcomputation, p checks its parent. If p's parent is a leaf, then work on the parent immediately. Otherwise, start work stealing.

The above requirement guarantees that there is no explosion in space, since it maintains a bound on the number of leaves. Note that other multithreaded systems do not necessarily guarantee the above space bound.

Example: Prove a tighter bound on S_P for a divide-and-conquer $n \times n$ matrix multiplication algorithm. Recall:

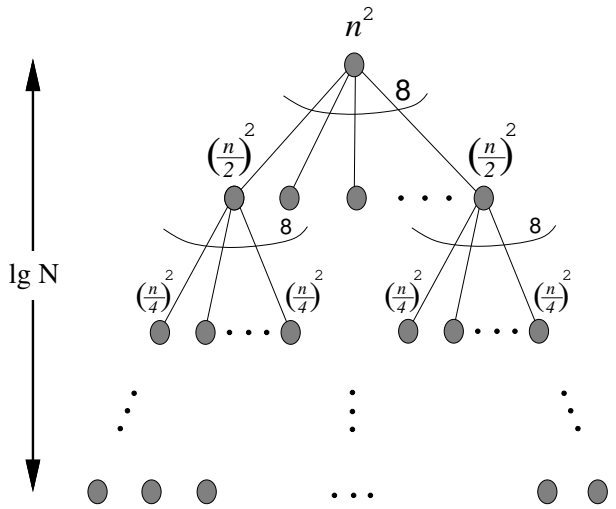


Figure 1: Space Requirement for Spawning Multiplies on an infinite number of processors

$$\begin{array}{lcll} \text{Work} & = & T_1(n) & = & \Theta(n^3) \\ \text{CP} & = & T_\infty(n) & = & \Theta(\log^2 n) \end{array}$$

Most of the space is taken by the spawns of the multiplies. Figure 1 illustrates the space requirements of the divide-and-conquer algorithm.

Then, the space required by a single processor:

$$\begin{aligned} S_1(n) &= S_1\left(\frac{n}{2}\right) + n^2 \\ &= \Theta(n^2) \end{aligned}$$

Weak bound on the amount of space utilized by P processors (due to busy-leaves scheduling): $S_P = O(P \cdot n^2)$. For infinite processors,

$$\begin{aligned} S_\infty(n) &= 8S_\infty\left(\frac{n}{2}\right) + n^2 \\ &= \Theta(n^3) \end{aligned}$$

We now prove a tighter bound on the amount of space by analyzing the worst case. As shown in Figure 2, the worst case execution tree involves a fully parallel top portion of the tree until there it reaches a level with P processors, and a serial portion from each of these processors down to the leaves. The space requirements of the transition level between the two tree portion dominates. Thus, we get an overall space requirement of $\Theta(n^2 P^{\frac{1}{3}})$.

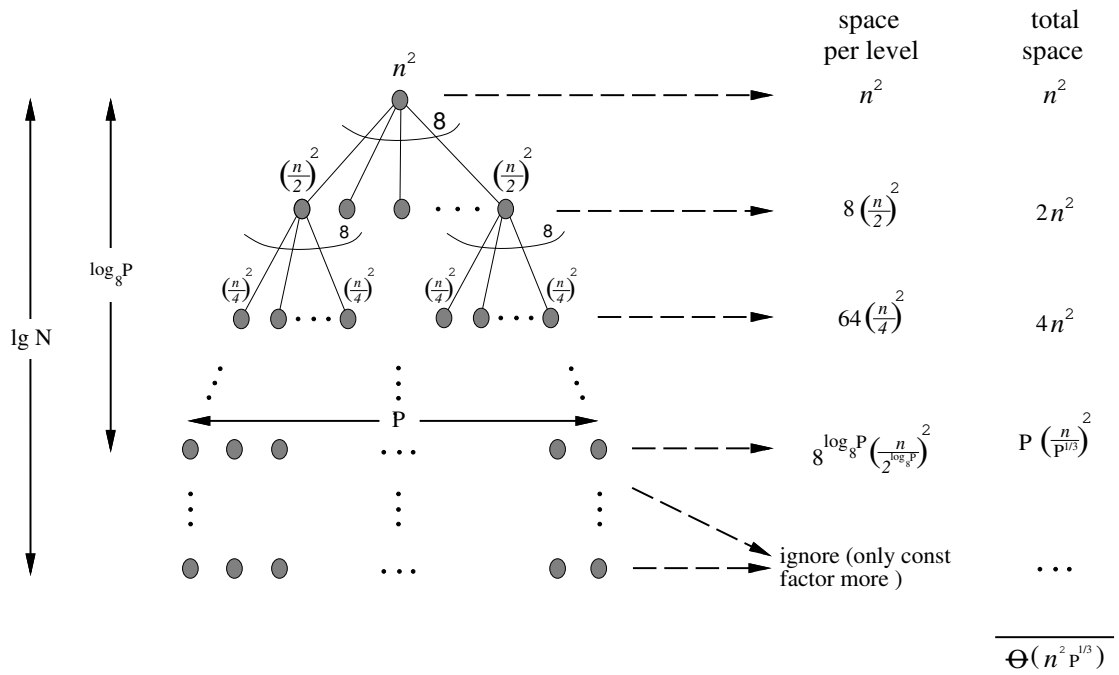


Figure 2: Worst Case Space Requirement for Matrix Multiply

General Theorem for "Regular" Divide and Conquer Algorithms

Definition: A "regular" algorithm obeys the following structure:

To solve a problem of size n:

```
{
  if n ≤ const, solve and return
  else
     allocate s(n) stack space
     divide problem into a sub-problems of size n/b each (a > 1, b > 1)
     spawn solutions of the a sub-problems recursively
     sync
     merge results
     return
}
```

On P processors, space S_P satisfies:

1. If $s(n) = \Theta(\log^k n)$, then $S_P(n) = \Theta\left(P \cdot \log^{k+1}\left(\frac{n}{P^{\frac{1}{\log_b a}}}\right)\right)$. [Note: There is a bug in the paper. This is correct]

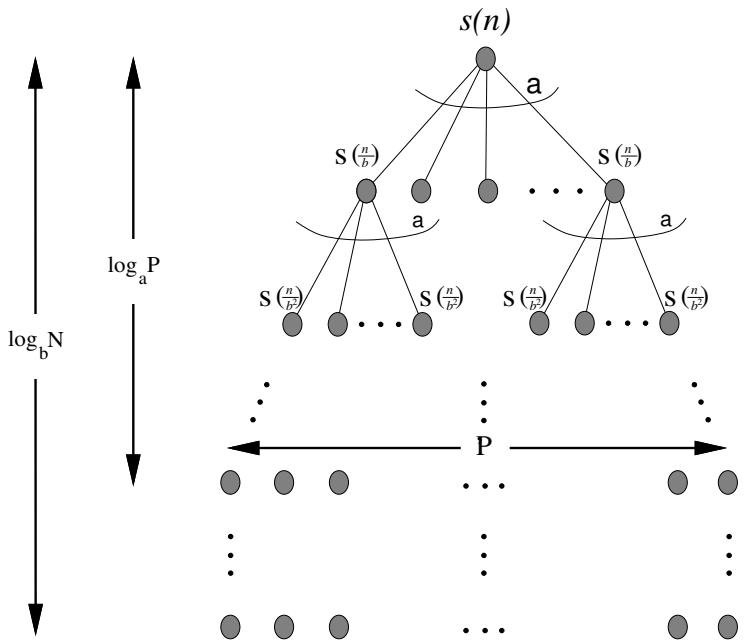


Figure 3: General Space Requirement for Divide and Conquer

2. If $s(n) = O(n^{\log_b a - \epsilon})$ for constant $\epsilon > 0$ [1], then $S_P(n) = \Theta\left(P \cdot s\left(\frac{n}{P^{\frac{1}{\log_b a}}}\right)\right)$.
3. If $s(n) = \Theta(n^{\log_b a})$, then $S_P(n) = \Theta(s(n) \cdot \log P)$.
4. If $s(n) = \Omega(n^{\log_b a + \epsilon})$ for constant $\epsilon > 0$ [1], then $S_P = O(s(n))$.

The proof for the above can be found elsewhere. Here we only provide the intuition behind the rules, as illustrated by Figure 3:

$$S_P(n) \leq \sum_{i=0}^{\log_a P} a^i s(\tfrac{n}{b^i}) + P \cdot S_1\left(\frac{n}{P^{\frac{1}{\log_b a}}}\right)$$

The first term above accounts for the top portion of the tree, while the second term accounts for the bottom portion of the tree. Notice that the computation of the bottom portion uses $\frac{n}{b^{\log_a P}} = P^{\frac{1}{\log_b a}}$. Then,

$$S_1(n) = S_1(\tfrac{n}{b}) + s(n)$$

Recall from the Master Theorem:

- $s(n) = \Theta(\log^k n) \Rightarrow S_1(n) = \Theta(\log^{k+1} n)$

[1] and $s(n)$ satifies the regularity condition

- $s(n) = \Omega(n^\epsilon)$ for constant $\epsilon > 0$ [2] $\Rightarrow S_1(n) = \Theta(s(n))$

[Note: $s(n)$ is regular *iff* in all ranges, $s(n)$ grows polynomially.]

The analysis consists of the following cases:

Case 1 Top of the tree increases geometrically from the root. The bottom decreases arithmetically ($k = 0$) and dominates.

Case 2 Top of the tree increases geometrically from the root. Bottom of the tree decreases geometrically and dominates (specifically, the last level in the top dominates).

Case 3 Top has the same space requirement at each level, and therefore dominates.

Case 4 The root dominates because the top decreases geometrically.

2 Worst Case Space Analysis

Here we consider only the worst case behavior for a restricted domain. First, let a *simple multithreaded algorithm* be an algorithm that in order:

allocates memory

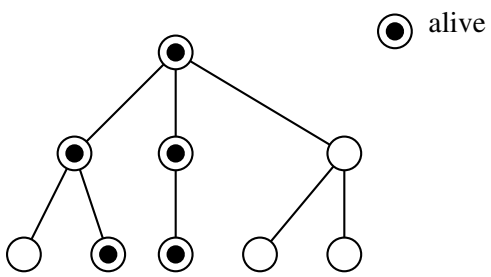
spawns and computes some number of times (≥ 0)

syncronizes

deallocates the memory

returns

Then consider the *alive subtree* of an execution tree, as illusrated here:



At any point in the execution, some nodes are *alive*, i.e. they have allocated memory, but have not deallocated it yet. Since the parent of an alive node is alive, the alive subtree is indeed a subtree of the spawn tree. During the execution of a Cilk program, we can

[2] and $s(n)$ satifies the regularity condition

observe the scheduling activity in time, and notice many alive subtrees. However, an alive subtree has at most P leaves (where P is the number of processors) because the scheduler is busy-leaves.

Then, to compute the overall space, let $s(v)$ be the space allocated by a procedure/node v. The total space used by an alive tree T is the sum of $s(v)$ for all nodes $v \in T$:

$$S(T) = \sum_{v \in T} s(v)$$

Then, the space S_P is at most the maximum of $S(T)$ over all trees, which have at most P leaves. Let T^* be any subtree that achieves the worst case. First only consider a special case of the regular spawn tree where the amount of allocated memory only depends on the level. Let $s(l)$ be the space allocated at level l. Assume that $s(l) > 0$ (e.g. space for the return address, or simply add some ϵ to the space requirement). Also, without loss of generality, suppose:

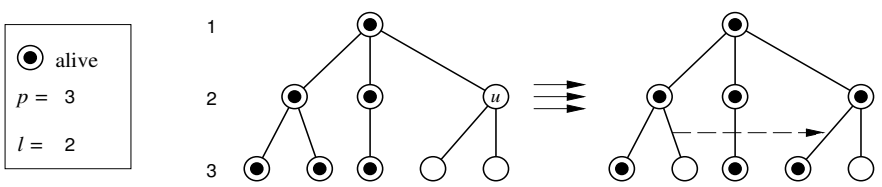
- The spawn tree has at least P leaves (otherwise, T^* is the entire tree).
- T^* has P leaves (otherwise, simply add another leaf to get something bigger).

From the above, we can prove the following theorem:

Theorem:
(1) *If there are at least P nodes at level l in the spawn tree, then there are P nodes at level l in T^*.*
(2) *If there are less than P nodes at level l in the spawn tree, then all of them are in T^*.*

Proof: Prove the following equivalent statement – "If there is a node in the spawn tree but not in T^*, then T^* has P nodes at the same level." (The proof mechanism is to use the *pigeonhole principle*[3]).

By contradiction. Let a node u at level l be in the spawn tree but not in T^*, and let level l of T^* have less than P nodes. There are P leaves in T^*, and therefore one node at level l is an ancestor of two leaves. Then, we can construct T' that has at least space $S(T^*) + s(l)$. An example is illustrated in the figure below. We attach the leaf to u together with the path. Notice that since this problem is "regular", node u must have a path that leads to a leaf node at the lowest level of the tree. The path we removed is at least compenstated by the path we add, plus the addition of u. Q.E.D.



[3]If you have more than P pigeons and only P holes, then at least one hole must contain 2 more or pigeons.

General Case

[1] shows how to handle the general case of simple algorithms for non-regular trees. The same theorem also applies for the case of non-simple algorithms, but may not be tight enough. Handling non-simple algorithms is still an open question. The following issues arise when we remove the restriction of a regular tree, and that s is a function of the level:

- It is better to associate a function $S(u)$ with a node u that provides the "sequential space" used by the corresponding subtree. The problem is that we cannot "pair up" nodes on the same level, and then need a measure of "how much worst case" we can gain by adding a certain node (with its corresponding sequential path).
- It is not clear when a level reallyis. We have only been able to define a level in the alive tree, not in the spawn tree. The following definition may be suitable: given a node u in an alive tree T, the "level" of u is the set of nodes $v \in T$ such that $S(v) \geq S(u)$, and v has a child w such that $S(w) < S(u)$.
- Observe that an alive tree is determined by the leaves. [1] considers the set of leaves Λ and denote the alive tree with those leaves with $\mathcal{C}(\Lambda)$.
- It is not clear that a level dominates all the leaves. A leaf may be a child of node node in a level, and this would destroy the above proof. We need to prove that this does not take place.

References

[1] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eigth Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, June 1996.